Approaches to automatic differentiation

Bart van Merriënboer
Google Brain and MILA

Automatic differentiation

Gradients

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

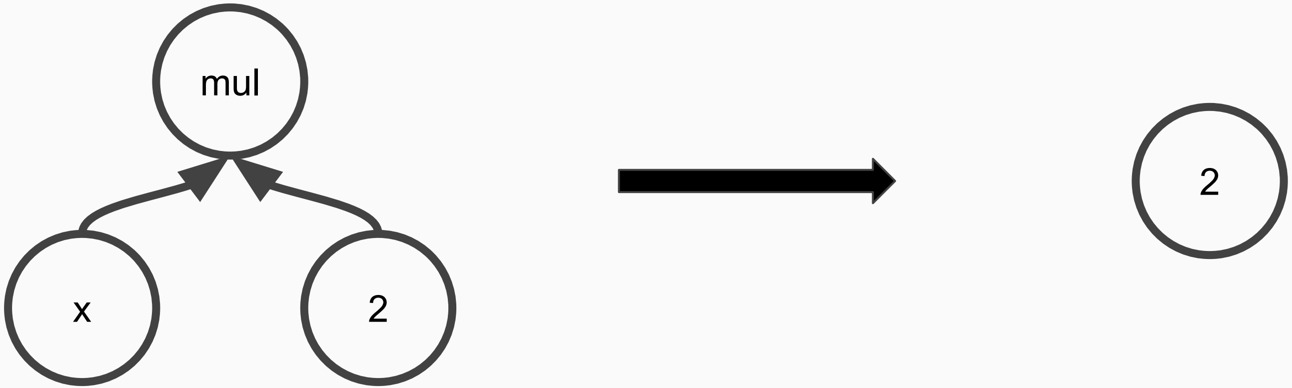
$$\frac{\mathrm{d}}{\mathrm{d}x}\left(f(g(x), h(x))\right) = \frac{\partial f}{\partial g}\frac{\mathrm{d}g}{\mathrm{d}x} + \frac{\partial f}{\partial h}\frac{\mathrm{d}h}{\mathrm{d}x}$$

~~Automatic~~ *Numerical* differentiation

Only the original function is needed.
Note that finite differences are an *approximation*.

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(a+\epsilon) - f(a)}{\epsilon}$$

~~Automatic~~ *Symbolic* differentiation



Automatic differentiation

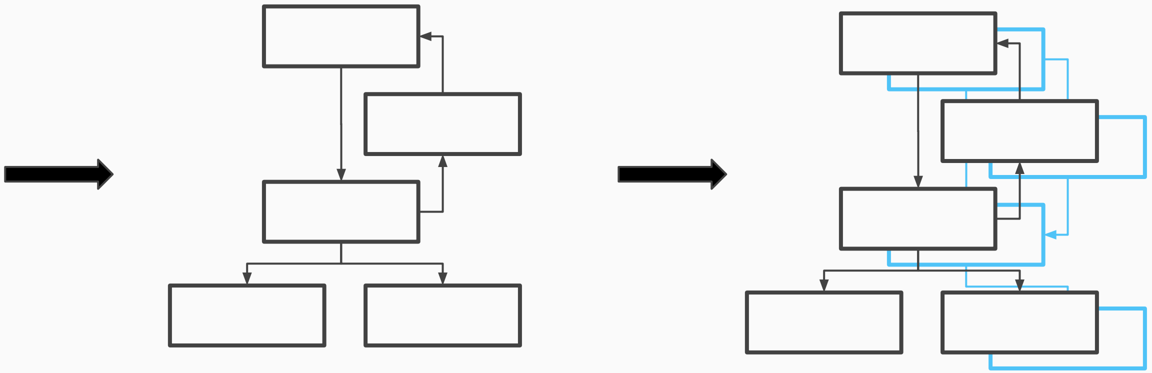
Automatic differentiation (AD) *[…] is a set of techniques to numerically evaluate the derivative of a* ***function specified by a computer program****. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of* ***elementary arithmetic operations*** *(addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the* ***chain rule*** *repeatedly to these operations, derivatives of arbitrary order can be computed automatically,* ***accurately to working precision****, and using at most* ***a small constant factor more arithmetic operations than the original program****.*

—Wikipedia

Automatic differentiation

```
def f(x):
    a = x * x
    b = log(a)
    return b

df = grad(f)
```



Automatic differentiation

- What program representation do we transform?
- Do we perform the transformation ahead-of-time (source code transformation) or at runtime (operator overloading)?
- How do we ensure that the transformed program is still amenable to efficient execution and compilation?

- How can the user debug the generated adjoint code?
- How can the user modify the generated adjoint code?

ML frameworks with AD support

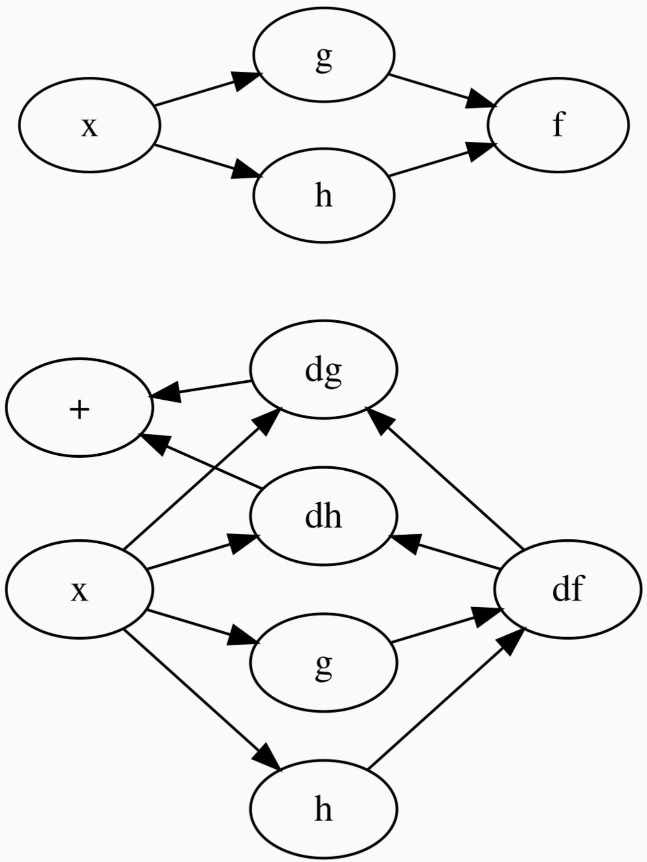
TensorFlow

- Python (or another language) is used to metaprogram a computation graph. This graph is transformed and executed with a custom pipeline.

```
x = tf.placeholder(tf.float32)
i = tf.constant(0)
c = lambda i: tf.less(i, 10)
b = lambda i, x: tf.add(i, 1), tf.tanh(x)
r = tf.while_loop(c, b, [i, x])
dx = tf.gradients(r[1], x)
```

Computation graphs

- Inspired from computer algebra systems and dataflow programming
- Allow the user to build a directed acyclic graph (DAG) where the nodes are functions and the edges are dependencies
- The graph is transformed into a new graph which calculates the gradient
- Example of $\nabla f(g(x), h(x))$



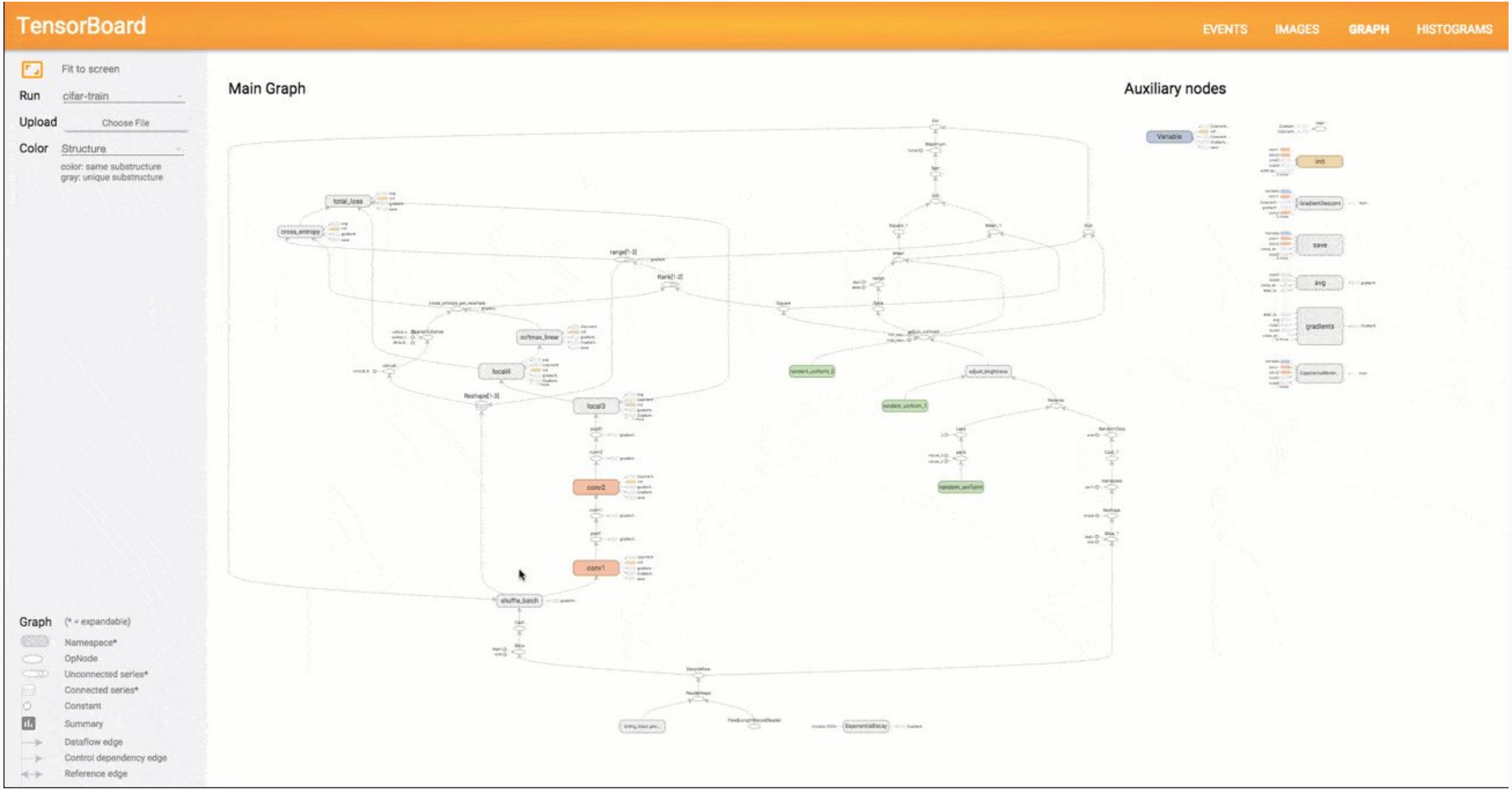
TensorFlow

Advantages

- Computation graphs are purely functional program representations without scoping, which makes them easy to transform
- Computation graphs and their gradient graphs are high level and can be manually inspected
- The two-stage execution model frees us from the Python interpreter (e.g. mobile deployment, XLA)

Disadvantages

- Metaprogramming introduces cognitive overhead, leads to verbose code, and requires two debuggers, two runtimes, two “languages”, etc.
- The limited representational power of computation graphs can complicate the implementation of some algorithms (e.g. those using recursion)



TensorBoard: Visualizing and inspecting computation graphs

PyTorch

Use operator overloading to trace the execution a Python program. Then transform this linear trace of computation.

```
x = torch.tensor(1, requires_grad=True)
i = 0
while i < 10:
    x = torch.tanh(x)
    i += 1
x.backward()
dx = x.grad
```

PyTorch

Advantages

- No metaprogramming required: More natural code which can include high-level programming constructs such as recursion and closures.
- Execution happens within Python (kind of)

Disadvantages

- Runtime overhead because of tracing through operator overloading
- Gradient code only exists as a data structure (linear trace) which is interpreted, can be hard to debug
- Execution happens within Python

Tangent

- Transform Python's AST directly and generate new source code

```
def f(x):
  a = x * x
  b = log(a)
  return b

df = grad(f)
```

```
def dfdx(x, init_grad=1.0):
    # Set the initial gradient
    db = init_grad
    a = x * x

    # Grad of: b = log(a)
    da = db / a

    # Grad of: a = x * x
    _dx2 = tangent.unbroadcast(da * x, x)
    dx = tangent.unbroadcast(da * x, x)
    dx = tangent.add_grad(dx, _dx2)
    return dx
```

Tangent

Advantages

- Human-readable source code
- Separation of concerns, integrates with the Python ecosystem: Step through your program with pdb, compile the code with Numba, etc.

Disadvantages

- Only runs in Python
- SCT is hard to implement for dynamic languages (needs mini Python compiler)

Other approaches

- Myia
  - Combine dataflow programming with functional language compiler representations to provide flexibility and high performance
  - Avoid metaprogramming by compiling a subset of Python
- Swift for TensorFlow
  - Build first-class AD support into the language's compiler
- JuliaDiff
  - First-class AD support for Julia

Take-home messages

- Automatic differentiation cannot be an afterthought; it impacts the entire development cycle of machine learning models
- Different implementations of automatic differentiation come with different trade-offs (ease of implementation, performance, usability, flexibility)
- Still work to do:
  - Languages with first-class AD support (research ones exist: VLAD, DVL)
  - Debuggers that understand the relationship between original and gradient code
  - Bring together writing kernels and models in a single framework

Thank you for listening. Questions?